S A L T · L A K E · C I T Y

# Summer 1984 Addendum to the Conference Proceedings

USENIX

Addendum to Summer 1984 USENIX Proceedings

# The Standardization of the C Language

**Lawrence Rosler**
**AT&T Bell Laboratories**
**Summit, New Jersey**

## Workshop Abstract

The American National Standards Institute (ANSI) Technical Committee for the Programming Language C (X3J11) has been meeting quarterly since June, 1983. The Committee is composed of more than 30 representatives of the C community, including vendors of compilers and applications, consultants and educators. The implementations considered run under the UNIX[1] System, under other systems, or stand-alone.

The goal of the Committee is to produce a draft standard for public comment early in 1985. Work is proceeding in three areas: environment, library, and language. The environment activity is attempting to standardize the characteristics of the compilation and execution environments, including the attributes of external files. The library activity is adapting the proposed /usr/group standard to the requirements of execution environments that are not UNIX-like. This session will focus on the language activity.

The base document for the language proper is derived from the C Reference Manual that is Appendix A of the 1978 book by B.W. Kernighan and D.M. Ritchie (K&R). The style and approach are fundamentally unchanged. Some copies of the most recent working draft will be available.

The proposed changes from K&R will be presented. Considerable time will be available for detailed discussion. Comments and suggestions from the audience will be appreciated.

[1]UNIX is a trademark of AT&T Bell Laboratories.

# A Simple Simulation Toolkit in C

Robert P. Warnock, III

Fortune Systems Corporation
Redwood City, California 94061

Bakul Shah

458 Ferne
Palo Alto, CA 94306

## ABSTRACT

A simple set of simulation tools is presented which offers much of the power of specialized simulation languages such as GPSS or SIMSCRIPT, while not interfering with the C programmer's native style. The toolkit, a library of C routines, comprises three conceptual layers (the user's simulation is the fourth), each reflecting some well-known design borrowed from other systems. From the "bottom" up, they are: (1) Co-routines and process objects; (2) Condition variables and priority queues; and, (3) A simulation kernel (the "clock" object and the "busy" statement). Co-routines and process objects are as in the BLISS language. Condition variables, priority queues, and the simulation kernel itself are from Holt's Concurrent Euclid. The overall idea of providing a simple skeleton for user-written simulations is from CMU's POOMAS package (POOr MAn's Simula).

Two examples of use are given, with different styles of output. One produces a discrete "event log" report, and the other a quasi-continuous "logic analyzer" display.

It is hoped that this paper will encourage programmers and engineers to make more use of simulations in their designs, as they realize that there is no particular magic involved.

## Introduction

From time to time, the professional programmer or engineer encounters design situations which may be too complex to handle easily with numerical or algebraic analysis. This difficulty may be due to theoretical intractability of the subject area, or it may be simply that the area is new to the designer, who does not yet have a reliable intuitive model to lean on in deriving his or her analysis. In either case, discrete simulation is an often useful tool to explore the design space and to gain some feeling for the magnitudes of the interactions between variables. In extreme cases, a simulation is the _only_ way to get usable numerical results. Even when acceptably accurate analytic results are available for overall system performance, simulation may reveal some surprising interactions which were unanticipated by analysis. (An example is catastrophic buffer congestion in packet networks.)

Simulation and analysis therefore can complement each other in refining a design. However, to many programmers and engineers, the term "simulation" conjures visions of large mainframes spending hours of CPU time, arcane unapproachable special-purpose languages, and many heroic hours of data and control-file preparation. For many of the simulation needs encountered in the microprocessor and minicomputer worlds, doing useful simulations need not involve leaving ones normal computing environment.

What is needed is a view of simulations as a normal part of ones personal skillset. Working through some introductory examples often helps one acquire this view, so after presenting the simulation toolkit, we sketch a modest DMA/memory-bus simulation to illustrate the major points. While quite simple, this simulation was useful in discovering a bus-arbitration problem in a commercial computer. The original version of the simulation toolkit and the DMA/bus simulation were developed over one "long weekend" while investigating this problem. It is hoped that this paper will encourage other programmers and engineers to make use of simulations in their designs, as they realize that there is no particular magic involved.

## Overview of the Simulation Toolkit

We describe a simple set of simulation tools which offer much of the power of specialized simulation languages such as GPSS or SIMSCRIPT, while not interfering with the C programmer's native style. In addition, it allows some experimentation with object-oriented programming techniques. Its features include:

Processes: Dynamic creation and destruction of any number of process objects or types; No process object semantics are assumed, only the bare framework is provided; Process objects may be passed parameters at creation time; Processes can call any normal C library subroutines (such as "printf").

Efficient: Reasonably low simulation overhead (medium-sized simulations run at human reading speed), process-switch time is comparable to subroutine call/return times.

Portable: Machine independent above the bottom layer (implementations for the bottom layer are available for the Motorola 68000 and the VAX 11/780).

Open: Any portion of the toolkit may be replaced or modified to fit a particular application. As discussed below, no output procedures are supplied. The user has complete control of the display.

The overall idea of providing a simple skeleton for user-written simulations is from the POOMAS package (Poor Man's Simula) [DECUS]. Written in the BLISS programming language (a goto-less type-less language of the ALGOL and BCPL traditions) [WULF71], POOMAS was a package of macros and library routines which allowed one to write BLISS programs in the style of SIMULA simulations [DAHL72b] [PALME73].

The approach discussed here, a library of C routines, is presented as three conceptual layers, while the user's application forms the fourth and fifth. Each layer reflects some well-known design borrowed from other systems. From the "bottom" up, they are:

1. Co-routines and process objects;

2. Condition variables and priority queues;

3. A simulation kernel (the "clock" object and the "busy" statement).

To create a complete simulation, the user supplies:

4. The output display and monitoring procedures; and

5. The procedures which model the simulated objects themselves.

## Layer 1: Co-routines and process objects

In order to simulate the behaviour of many objects acting "simultaneously", we model each object as an independent process. Processes communicate via global variables or pointers to shared data objects. For efficiency, all of the simulation processes are sub-processes of one timesharing process or job [1], effectively providing multiprogramming within the job. The lowest level of the toolkit is therefore a process scheduler, which allows individual sub-processes to be suspended and resumed later. This can be easily done in any programming language which provides a mechanism for coroutines.

Coroutines [KNUTH73] [DAHL72a] are a generalization of the more familiar subroutine mechanism, in that when one coroutine A "returns to" or "calls" another coroutine B, the location of the call in A is remembered. When A is called again (by B or other coroutines), it resumes from where it left off, rather than starting over at the top. Where coroutines are not supported by standard software, they can often be provided by a small assembly-language routine [2].

The co-routine and process object models used here are derived from

[1] To simplify terminology, from here on "process" will mean a sub-process within one timesharing process, job, or program. Note that a "job" is a single UNIX user process.

[2] The C library routines "setjmp(3)" and "longjmp(3)" may be used to implement coroutines. We chose to provide a single assembly-language "resume", for clarity. See also "ctxsw.s", in [COMER84].

the exchj (exchange jump) and create constructs in the BLISS-10 language [DEC73]. A process object or process is defined by a data structure which contains "static" variables local to the process, a stack of subroutine call frames, and the process context (PC, stack pointer, registers) when not active. To the C compiler or the operating system, a process object is merely another user data space object allocated either statically from a fixed array or dynamically from the heap via "malloc".

At any one time, exactly one process is running, and the global variable co currentproc holds a pointer to it. The state of the current process may be saved and another process activated with the call:

```
                    co_resume(other_proc);
```

This is a very low-level mechanism, and does not save the address of the current process object, as "co_currentproc" is overwritten with a pointer to the new "other_proc". The caller must arrange for the current process to be enqueued (or otherwise remembered) for subsequent activation, or the "co_resume" call can never return. (Normally, the caller is one of the Layer 2 or Layer 3 routines which perform exactly this function.)

But before a process can be run with "co_resume", a process object must be created for it, and the object must be filled in with a valid stack frame, address of the code for the process, etc.[3] This is done by allocating space [4] for the process object, then calling co create:

```
         proc = co_create(address, size, handler,
                          function, arg1, arg2, ..., argN);
```

This creates a process object in the space given by address and size.

Function is the code which gets executed when the process is resumed, and is an ordinary C subroutine. The stack frame is arranged so that on the first resume of the process, the function is entered from the beginning exactly as if it had been called with a normal subroutine call with arguments arg1 ... argN. The return value of "co_create" is a pointer to the process object, for use in later calls such as "co_resume".

If the function should exit via a normal return, handler is called (by the internal routine co exit ) with arguments of the address of the process object, its size, and the return value of "function". The exit handler should perform whatever cleanup is needed (such as de-

[3] A small dummy process object must also be created and initialized for the main routine before the first "co_resume" in the program.

[4] Choosing the size may be tricky. See Appendix A.

allocating the space for the process object), and resume some other process. It is a fatal error for the exit handler to return.

## Layer 2: Condition Variables and Priority Queues

After arranging for many processes and a way to switch between them, the next thing needed is a method of synchronizing their executions. For that, we use condition variables and priority queues as in Concurrent Euclid [HOLT83], however "monitors" per se are not provided. Rather, the condition queues are used directly by the user for interprocess synchronization [5].

A condition variable is a queue of processes waiting for the condition associated with that variable to become true. A process which executes the call

```
q_wait(cond)
```

waits on the condition queue for cond (which must have been declared q var). A later signal call will remove the first such waiting process. By giving a second argument,

```
q_wait(cond, prio)
```

the process waits on the condition queue for cond with priority prio (lower integer values are higher priorities). A later signal will remove the process with highest priority.

```
q_signal(cond)
```

If there is a process waiting on cond, the signal puts the current process onto the ready queue, and resumes the highest priority process waiting on the condition. If there are no processes currently waiting, it simply selects the next process from the ready queue and resumes it (possibly the same one which called "signal").

## Layer 3: Simulation kernel

Once you can switch between processes and can synchronize them with condition queues, the rest is easy. The basic function which turns all of the above mechanism into a simulation package is the busy statement.

```
sim_busy(delta_time)
```

This puts the current process on the sleep queue so that it will run

[5] On a single-processor system without interrupts, this is not a limitation. True monitors can be implmented, if needed, by adding the routines "mon_enter()" and "mon_exit()".

after delta time units of simulated time have elapsed, and selects another process for running.

When there are no available processes to run on the ready queue, simulated time (the global clock variable sim time ) is advanced to the time of the next event on the sleep queue, and that process is dequeued and run. This has the following implications:

1. Simulated time advances in "leaps", skipping over periods during which nothing happens. This helps to make the simulation reasonably efficient.

2. No matter how much real CPU time a process takes, it will appear infinitely fast in the simulation. The "busy" statement and the "wait" are the only ways simulated time can pass.

Finally, to get a process going in the beginning, there is the start statement.

```
        sim_start(proc)
```

This starts process proc immediately by placing it on the ready queue (although it will not actually run until the current process and all other ready processes have executed "busy" or "wait" statements).

## Writing Simulations Using the Toolkit

To construct a complete simulation, the programmer must add two more "layers". The first is the output display routine or process, and the second comprises the actual routines to define the behavior of the objects in the simulation. While conceptually the output function is a "lower" level than the simulation itself, in fact they must be designed together. The selection of what portion of the state of the simulation to display and how to format it is very dependent on the application domain and on the properties of the simulated system.

Simulations are usually concerned with a variable number of objects which arrive (appear), spend most of their lives waiting in one or more queues to get service, get some service or other (possibly looping in the system), and eventually depart (disappear). Sometimes the number of objects is fixed, and what is being simulated is various state changes within the objects as they receive simulated input from outside the system and interact with one another.

In most cases, however, the information desired from the simulation will fall into one of two general styles, which we will call event-oriented or time-oriented depending on the style of the output display. Oddly enough, the output style seems to have little to do with whether the system has a variable number of objects or a fixed number, but whether one is focussing on the fine-grain sequential behavior or the overall parallel behavior of the system.

The event-oriented output display shows each of the major transitions of each object, queue, or server, like this example:

```
Time(ns): Device: Event
=================================
       1: clock: signal busFree
       1: CPU: bus acquired
      10: ENET1: next word ready
      10: ENET2: next word ready
      10: DISK: next word ready
    1610: ENET1: next word ready
    1610: ENET2: next word ready
    1621: CPU: bus released
    1621: REFRESH: bus acquired
    2161: REFRESH: bus released
    2161: ENET1: bus acquired
    2701: ENET1: word transferred
    3210: DISK: next word ready
    3210: ENET1: next word ready
    3210: ENET2: next word ready
    3241: ENET1: word transferred
    3781: ENET1: word transferred
    3781: ENET1: bus released
    3781: ENET2: bus acquired
    4321: ENET2: word transferred
```

This mode is generally implemented by display calls inside each queue or server object. These may be as simple as a "printf()", but for ease of modification it is more usual to have a common print routine, which may choose to output or not depending on some global printing-level parameter (much like "DEBUG()" lines used in many C programs).

When one is more interested in the overall state of the system, such as the size of queues or the state of several objects simultaneously, it is often convenient to view time as quasi-continuous. The time-oriented or "logic analyzer" style of display shows a fixed amount of time per line of output, and each line outputs the status of a fixed number of objects (like the "channels" of a logic analyzer). An example:

```
Time(ns) BUS     Refresh Enet#1  Enet#2  Disk    CPU     Bus clock
=================================================================
  14590 ENET1           DMAread DataRdy                 up  \\
  15130 ENET2                   DMAread                 up  \\
  15670 CPU                                     I/Oread up  \\
  16210 CPU             DataRdy DataRdy DataRdy I/Oread up  \\
  16750 CPU     RefrReq DataRdy DataRdy DataRdy I/Oread up  \\
  17290 REFRESH MemRefr DataRdy DataRdy DataRdy         up  \\
  17830 ENET1           DMAread DataRdy DataRdy         up  \\
  18370 ENET1           DMAread DataRdy DataRdy         up  \\
  18910 ENET2                   DMAread DataRdy         up  \\
  19450 ENET2           DataRdy DMAread DataRdy         up  \\
  19990 ENET1           DMAread <wait>  DataRdy         up  \\
  20530 ENET2                   DMAread DataRdy         up  \\
  21070 ENET2           DataRdy DMAread DataRdy         up  \\
  21610 ENET1           DMAread         DataRdy         up  \\
  22150 DISK                            DMAwrit         up  \\
  22690 ENET1            DMAread DataRdy <wait>          up  \\
  23230 ENET2                    DMAread <wait>          up  \\
  23770 DISK                             DMAwrit         up  \\
```

For this mode of display, the objects must record their states in globally accessible locations, so that an independent display process can sample those states at regular intervals to format and output them. (To truly emulate a logic analyzer, one may even keep a pair of "transition detectors" for each state variable, as "glitch catchers", so that time intervals with more than one clean transition can be noted.) Time-oriented displays have the advantage that one may stretch or compress time without altering the simulated objects, merely by changing the loop time in the display process (note the fine detail now showing in the "bus clock"):

```
Time(ns) BUS     Refresh Enet#1  Enet#2  Disk    CPU     Bus clock
================================================================
14754 ENET1            DMAread DataRdy                    dn |
14784 ENET1            DMAread DataRdy                    up  \\
14814 ENET1            DMAread DataRdy                    up    |
14844 ENET1            DMAread DataRdy                    up    |
14874 ENET1            DMAread DataRdy                    dn  //
14904 ENET1            DMAread DataRdy                    dn |
14934 ENET1            DMAread DataRdy                    dn |
14964 ENET2                    DMAread                    up  \\
14988 ENET2                    DMAread                    up    |
```

The displays shown above were output from two versions of a DMA/memory-bus simulation used to investigate interactions among devices generating a heavy DMA bus load in a commercial microcomputer. The devices being emulated were the CPU itself (making continuous references to slow I/O registers), two 10 Mbit/sec Ethernet controllers, a 5 Mbit/sec disk controller, and the dynamic RAM refresh process. To get a feeling for an event-oriented process, consider the routine "refresh" which emulated the RAM refresh logic, shown in its entirety below:

```
refresh(name, busFree, busCycle, priority)
char * name;
q_var busFree;
int busCycle, priority;
{
    int refreshPeriod = 14580; /* ~2ms. / 128 rows */

    for (;;) {
        wait(busFree, priority);           /* grab the bus */
        msg(name, ": bus acquired\n");     /* got it */
        busy(busCycle);                    /* refresh one row */
        msg(name, ": bus released\n");
        signal(busFree);                   /* let go */
        busy(refreshPeriod);               /* wait a bit */
    }
}
```

Below is the same "for" loop for the time-oriented version of "refresh":

```
        for (;;) {
            msg2(name, "RefrReq");          /* say we need bus */
            wait(busFree, priority);        /* try to get it */
            msg2("BUSUSER",name);           /* got it! say so */
            msg2(name,"MemRefr");           /* in my column too */
            busy(busCycle);                 /* refresh one row */
            msg2("BUSUSER","         ");    /* say we're done */
            msg2(name,"         ");         /* both places */
            signal(busFree);                /* release bus */
            busy(refreshPeriod);            /* wait a while */
        }
```

While it is possible to intermingle both styles in one simulation, it gets quite cumbersome, and is not recommended. If it is absolutely required to have both, a more sophisticated display package can simplify the appearance of the process routines, but inevitably introduces its own run-time overhead.

## Summary

We have outlined a simple set of tools which provide the C programmer the ability to perform discrete event simulations. The toolkit uses coroutines, condition queues, and a simulated clock to provide quasi-parallel execution of multiple sub-processes in a single user program. An example was sketched of an application of this toolkit to the analysis of a computer's memory bus. We encourage others to experiment with the use of simulations in design analysis and problem solving.

## References

[COMER84] Douglas Comer, Operating System Design, The Xinu Approach, Prentice-Hall (1984), 59-61

[DAHL72a] Ole-Johan Dahl and C. A. R. Hoare, "Hierarchical Program Structures", in Dahl, Dijkstra, and Hoare, Structured Programming, Academic Press (1972), 184-193

[DAHL72b] ibid, 210-220

[DEC73] BLISS-10 Programmer's Reference Manual Version 3, Digital Equipment Corporation (1973), 45-49

[DECUS] Carnegie-Mellon University, "POOMAS: POOr Man's Simula", available through Digital Equipment Computer Users Society (DECUS) DECsystem-10 Library [incomplete reference]

[HOLT83] R. C. Holt, Concurrent Euclid, The Unix System, and Tunis Addison-Wesley (1983), 93-113

[KNUTH73] Donald E. Knuth, The Art of Computer Programming, Volume 1, "Fundamental Algorithms", Addison-Wesley (1973), 190-196

[PALME73] Jacob Palme, "A Simula System for the DEC-10 Computer", Report #C 8352-M3(E5), Swedish Research Institute of National Defense (1973)

[WULF71] William A. Wulf, D. B. Russell, and A. N. Habermann, "BLISS: A Language for Systems Programming", Comm. ACM, 14, 12 (Dec. 1971), 780-790

## Appendix A - Hints, Restrictions, and Warnings

### Stack Locations

The primary obstacle to this approach is hardware and/or software which is restrictive about where user stacks may be located, since the the process objects and therefore the stacks are kept in user data space. It is possible but cumbersome to construct a dynamic coroutine mechanism in which the stacks live within the job's main stack space. Fortunately, most UNIX* implementations permit stack pointers anywhere in user data space.

### Stack sizes

The user is responsible for determining the size of the process object, depending on the number of "static" variables and the size of the stack needed for the process. The main process is the only one for which system-provided stack growth facilities will work, so if memory space is scarce, the main process (once it has started the simulation) may best be used as a service process to handle stack-intensive requests (such as "printf(3)").

Although the tuning of the object size and watching for stack-overflow problems can be tricky for large simulations, in fact the authors have not encountered stack problems in the medium-sized simulations this toolkit was written to handle. Each process was simply given an ample stack (1000 bytes or so), and no fine adjustments were done.

### "Nargs"

The routine "nargs" is very machine dependent, and cannot be written at all on some machine or compiler implementations. Its main use here is in "createProc", where it provides a naturalness to the process procedure invocation. (A lesser use is in the "wait" routine.) The use of "nargs" can be avoided if necessary in several ways: (1) add a parameter to "createProc" to explicitly indicate the number of arguments, (2) force all processes to have a fixed number of arguments, (3) use an argument block pointer (such as "argc" and "argv") or (4) place the arguments into the process object at process-creation time.

*UNIX is a Trademark of Bell Laboratories.

# Relating Benchmarks to Performance Projections
or
What do you do with 20 pounds of benchmark data?

Gene Dronek

Aim Technology
Santa Clara, Ca 95051
(498) 727-3711

## 1.0 ABSTRACT

UNIX operating systems seem to invite benchmarking measurements. So much so, UNIX journals now include benchmark-of-the-month columns. But can you really use benchmarks to justify purchase decisions, or is benchmarking merely an interesting exercise?

At Aim Technology, we experimented with a linear-combination model that projects performance from 1 to 50 columns of raw benchmark data. Our stated goal was to construct a decision-making tool for use by non-experts. In our implementation, you request a projection by specifying percentage mixes of applications, such as 10% word-processing plus 90% compiling.

At first, organizing displays of benchmark database data in reasonable space was difficult. We tried Kiviat diagrams and found they could display 50 performance factors for 20 machines simultaneously on one diagram. Although artistic looking, Kiviat diagrams do not compare as easily as rectangular plots. Ultimately, Kiviat diagramming inspired the compact displays we call VIEWS.

In the paper, we describe the linear model in detail, with aspects of software engineering. We define the "Checksummed Header" puzzle for readers to enjoy before the conference.

## 1.1 PROJECT GOALS

Work on this project was done at Aim Technology in the ten month interval from August 1983 to May 1984. We had three stated goals. Primarily, we wanted a professional tool for computer-purchase decision making. It had to be utterly portable, and it was to be used by non-experts.

Secondarily, we wanted a tool that could also generate reports suitible for publication. Reports had to be field-configurable, again by non-experts.

## 2.0 BENCHMARKING PROCEDURE

The benchmarking strategy we developed separates data acquisition from data reporting. An ascii database connects the two phases. This arrangement fosters portability and allowed us initially to experiment with several data acquisition methods, yet utilize one common report generator. Early during development, we used twelve performance measuring programs, all feeding into the one database. Eventually, all twelve codes were integrated into one quick benchmark code -- dubbed "Q-BENCH" (for Quick-Benchmark), but the ascii database format was kept.

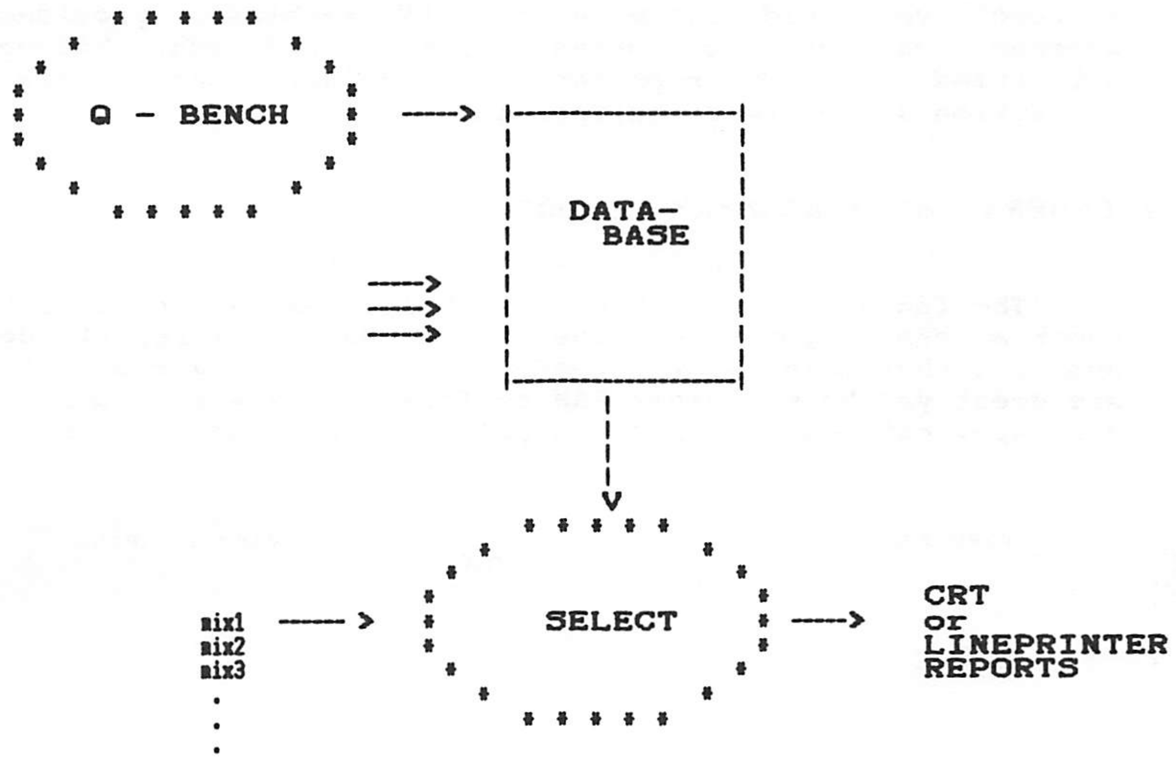


fig 1: Benchmark Data Flow

Presently, program Q-BENCH runs some 36 separate tests that measure individual performance capabilities (e.g. disk, ram, ...). At termination, Q-BENCH outputs one very long line of data which is appended onto the database. The line consists of a 20-character machine name, followed by 50 blank-separated numeric fields, each representing performance rates.

Generally speaking there are two kinds of benchmark data one could measure--intervals, and rates. Interval data is most common for benchmarking and extremely easy to generate using the UNIX "time" command. However, interval data is hard to compare, and suffers from the complication that "smaller" intervals result from "better" performances.

Q-BENCH exclusively determines "rates" of performance, where the work-per-second of some factor is measured. For example, disk performance is recorded in bytes-per-second, system call performance is measured in calls-per-second, and so forth. These rate measurements yield to handy mathematical manipulation as described later.

We purposely chose a simple format for the database so standard UNIX utilities could be used to manipulate it, and so the database itself could easily be carried from machine to machine. If necessary, to transport benchmark data, you could hand copy the numbers off a CRT and later use "echo" or "cat" to append it onto the database. Potentially, with this approach, you could benchmark non-UNIX machines, provided the Q-bench code can be gotten to compile and run, but we have not tried yet. Other portability considerations are discussed in section 4.0 Software Engineering.

## 2.1 GENERATING BENCHMARK REPORTS

The fun has only just begun. While a standard set of reports can be generated, the usefulness of the reports depends upon how they match your intended use for the systems. Chances are great you know whether RAM performance is more important for your applications, or FLOATing point, or DISK and so forth.

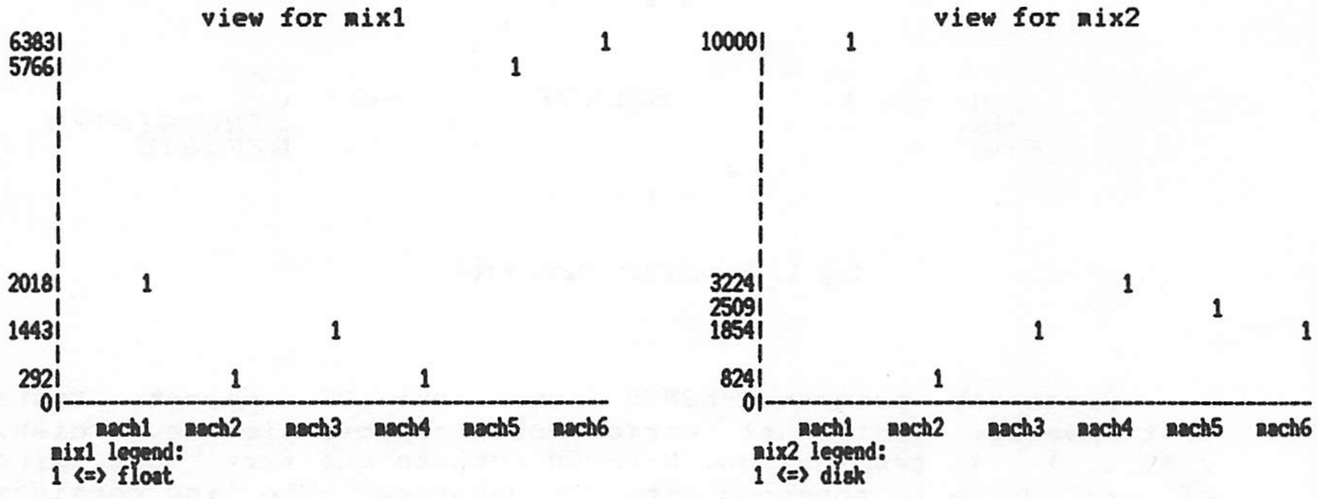
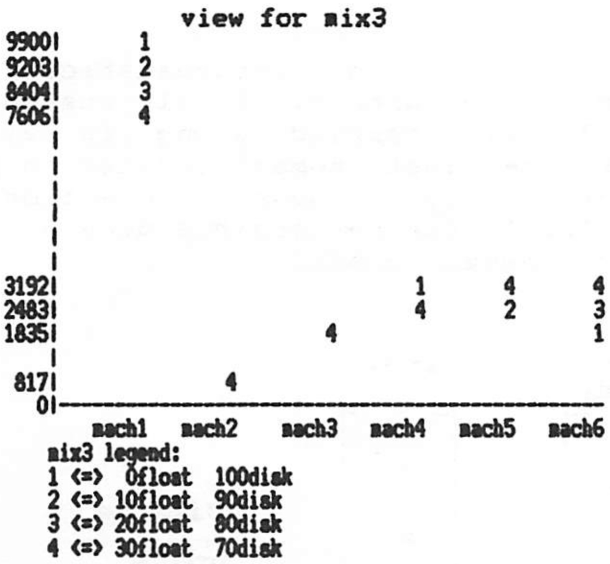


fig 2: Single-factor Projections

The primary display, called a VIEW, compares multiple-machine performances on a single chart. Two single-factor views are show in figure 2. Notice there is little consistency between floating-point performance and disk.

## 2.2 FORMING COMPOUND MIX EXPRESSIONS

In keeping with UNIX philosophy, we shunned interactive parameter entry, and instead built the report generator to read named files for the pre-rolled analysis formulas. The format is trivial, so your favorite editor can be used to create new mix files. For example, a mix file containing just "float", causes a report of those tests associated with floating-point performance. Similarly, "disk" causes a report of just disk performance tests. The report generator internally transforms each mnemonic mix name to a weight vector, by simple table lookup. A hidden text file (defs) contains mnemonic vector definitions. It too can be edited.



```
            view for mix3
9900|      1
9203|      2
8404|      3
7606|      4
    |
    |
    |
    |
    |
    |
3192|                                1       4       4
2483|                                4       2       3
1835|                        4                       1
    |
 817|              4
   0|-------------------------------------------------
       mach1   mach2    mach3   mach4   mach5   mach6
mix3 legend:
1 <=>  0float  100disk
2 <=> 10float  90disk
3 <=> 20float  80disk
4 <=> 30float  70disk
```

**Figures 2 and 3 are projected from the same database accumulated by benchmarking 6 machines with qbench. Machine identities have been removed. The vertical axis is normalized on a 10000 point scale, "up" corresponds to higher performance.**

**In figure 2, "mix1" compares floating point instruction speeds, and "mix2" compares disk transfer speeds. In figure 3, "mix2" considers 4 different application mixtures for various percentages of floating and disk. See text for explanation.**

fig 3: Two-Factor Projection

Mixes can be expressed as single factors, or as linear combinations of single factors. Thus complex analyses can be described and carried out. For example, you can devise a mix file which carefully varies a series of weights as show below.

```
 0float  100disk
10float   90disk
20float   80disk
30float   70disk
```

Notice how the smooth 10-percent steps examine different floating point and disk performance needs. Figure 3 reveals several up/down performance trends that can be picked out by inspection.

Regretably, some performance marks appear to be missing because the PLOT utility cannot display overlapping projections. Whenever performances pile onto the same PLOT point, only the higher-numbered label will be visible. Nevertheless, this lack of output resolution usually doesn't hide performance trends.

## 3.0 MATHEMATICAL BACKGROUND

The main obstacle to understanding and modelling computer performance is the inability to do so with single parameters. Although human perception demands it, there simply is no "single" figure of merit which will suffice all the time for all analyses. Computer performance is sufficiently complicated, that many components are needed to distinguish and quantify performance.

The way we handle this problem is not only programmatically easy, but central to the power of our method. Simply stated, all components, (50 at present), are carried along in all necessary calculations but at the last moment reduced to a single figure-of-merit ("points") by linear projection. Mathematically speaking, we multiply the performance data with the weight vector implied from the mnemonic mix.

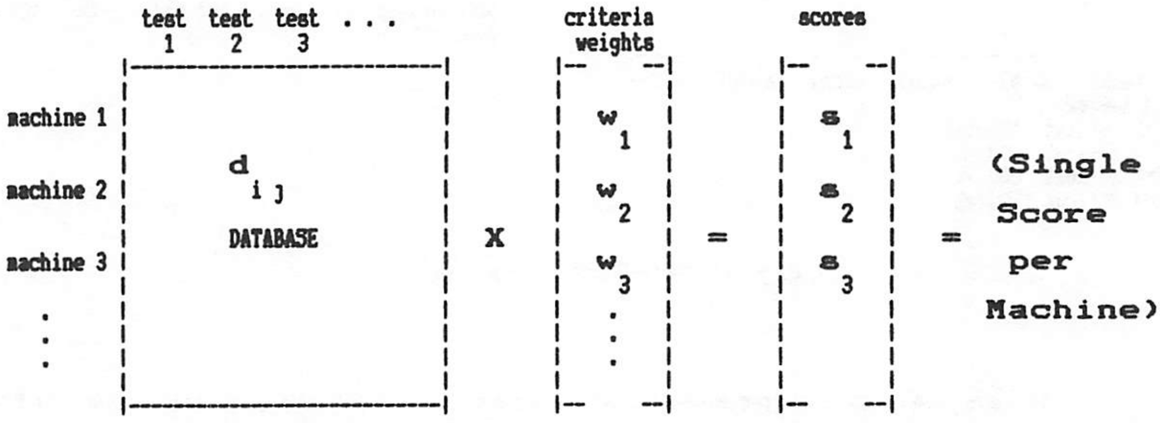


fig 4: Linear Projection

Projected scores from this method have several nice properties. The most favorable property is that of linearity. Twice the points corresponds to twice the performance; charts based on "points" can be directly compared. Furthermore, total-possible points is easily derived. In all our figure examples, the mnemonic defs file contained 100-point mnemonic definitions and with 100-percent weights this gives a final 10000 (=100X100) linear point scale. Finally, since scores are computed as linear combinations of linear combinations, ANY other analysis process consisting of linear combinations will be equivalent to our method.

## 3.1 GENERALIZED PROJECTIONS

The simple linear calculations we use allow a surprizingly powerful generalization for specific application needs. First, consider how application packages can be parameterized as weights. For example, to model word processing, you could say word processing needs good RAM performance, some PORT performance, and a little DISK performance, but no FLOATing point performance. Then, by treating "application" definitions, the same as performance factors, the same report generator program will perform application-oriented evaluation as well as factor-oriented evaluation. The overhead for this flexibility is just a few more lines in the mnemonic definitions file.

```
                 view for mix4
 7269|                                       1
 6940|                                       3
     |
 5628|    3
     |
     |
 4416|    2                         3
 3841|                  1           1
     |
 3140|                  2
 2476|    1
 2140|                  3      3
 1464|                         2
  738|           3             1
     |
    0|------------------------------------------
       mach1  mach2  mach3  mach4  mach5  mach6
    mix4 legend:
    1 <=>  10us 90sp
    2 <=>  50co 50sp
    3 <=> 100ac
```

fig 5: Appplication-Oriented Projection

Figure 5 shows a complicated what-if projection for three mixes running on the same machines from figures 2 and 3. It shows how the systems should perform for three application-oriented loads. Recall that the view is really just a mathematical projection, not an actual performance measurement. The first mixture line means ten-percent userfriendly services and ninety-percent spreadsheet, the second mix line means fifty-percent compiling and fifty-percent spreadsheet, and the final mix line means one hundred-percent accounting use. Values for the exact weights defined as userfriendly services (us), spreadsheet (sp), and so forth, are listed in the appendix. (They were voted upon by a committee of benchmark experts (friends) and left intentionally easy to edit to reflect personal tastes.)

Theoretically, there are an infinite number of ways to quantify application-oriented needs. Any two committees need not agree on the particular weights in question, but if they agree to use linear modelling, they can both use the same report generator. Our reporting method, which is subjective in just this one respect, nevertheless seems to produce consistent results. All other portions of our benchmarking and report generation method are objective, and mathematically justifiable.

## 4.0 SOFTWARE ENGINEERING

In retrospect, we can see how design decisions were influenced by the nature of UNIX. In shopping-list form, we took full advantage of many UNIX and C features: We used #define symbols to control different OS versions, #include files and V7 system calls for measurements, "make" to encapsulate benchmarking and reporting procedures, and shell scripts to remember option settings.

Certainly the Bourne shell had much impact, suggesting if you will, a preferred order of implementation. As shown in figure 6, there are three layers to be seen in the project software: Specialized C-code, general-purpose C-code, and specialized shellscripts.

| SPECIAL-PURPOSE C-PROGRAMS | GENERAL-PURPOSE C-PROGRAMS | SHELL-FILES and SCRIPTS |
|---|---|---|
| 1170 qbench.c | 138 cksum.c | 15 KIVIAT |
| 780 select.c | 563 plot.c | 15 POINTS |
| | | 23 VIEW |
| | | 46 defs |
| | | 24 makefile |
| | | 7 mix1 |
| | | 1 mix2 |
| | | 3 mix3 |
| | | 6 mix4 |
| | | 5 mix5 |
| | | 1 mix6 |
| | | 1 mix8 |
| | | 3 mix9 |
| 1949 lines | 699 lines | 150 lines |

--- order of implementation --->

fig 6: Software Design Layers

Following the "fail-early" principle (John Mashey-"Small is Beautiful" talk Dec 1982), we started coding the hardest programs first. The reasoning being it is more preferable to fail early than to fail late in a project. Secondly, in the event we ran out of time to develop everything as planned, we could skimp on the general C-code, and skip some shellscripts, but still have a working product.

Our final observation concerns the number of text lines within each layer. Each successive layer uses fewer lines. Presumably this is because each higher layer utilizes more "general" (and presumably familiar) programming concepts. Each layer certainly was easier to code, and it was most pleasant not to deal with prematurely-closed design decisions. It is all too common for the 10-percent remainder of a programming project to consume equally as much work as the first 90 percent. In our case, the final layer took about 2 days to develop and perfect, while the first layer took about 4 man-months. The middle layer programs were actually borrowed from another project.

## 4.1 SOFTWARE PORTABILITY ISSUES

The need for portability in benchmarking packages goes without saying. Benchmarks should run easily on a lot of different systems in order for them to have any chance of being used. In the name of "portability" we coded conservatively, and wrote all intermediate files in clear ascii text. The implicit relationships in our design from separating

mix-files and defs-files from the benchmark database encouraged conceptual portability as well.

Nevertheless, we ran into a few difficulties.

```
        *  clock/tick dependence
               50Hz, 60Hz clocks and 100:1 cpu range
        *  16-bit integer precision
               int x   ->   unsigned x
        *  compiler bugs
               a[a[a[a[a[a[a[a[a[a[a[a[a[a[a[a[i]]]]]]]]]]]]]]]]  ->
               a[a[a[a[i]]]], a[a[a[a[i]]]], a[a[a[a[i]]]], a[a[a[a[i]]]]
        *  crt/lineprinter resolution
               crt 80X24 and lineprinter 66X132
```

In order to sidestep a multitude of known weaknesses with UNIX clocks, we devised an auto-ranging strategy for taking measurements. The auto-ranging function runs each benchmark until a minimum reliable interval is exceeded. Cpu-tests are run at least 2 seconds. Disk and port tests run up to 30 seconds as we found this necessary to void buffer caching. To further increase reliablity, three distinct runs of the benchmark are averaged together internally. We found averaging three runs provides sufficient smothing effect for clock "noise". Actually, to top if off, we usually suggest running Q-bench twice on each machine, just to be sure.

We ran into difficulties on some very fast 16-bit machines, where the RAM tests needed all 16-bits to represent loop counts. Installing a few "unsigned" declarations fixed this problem. More treacherous was the 16-deep subscripting test, which made several compilers loop endlessly, and generate bad code. We simplified that test in the name of portability.

Finally, we made a few concessions for portability when designing report formats. We implemented "height" and "width" parameters in the report generator and set them via shellscripts so the user is unbothered by such details. This way, the scripts adapt to whether output is for CRT or lineprinter. Even so, some reports simply will never fit on a crt.

## 5.0 CONCLUSIONS AND COMMENTS

With respect to software design, we make two observations. First, separating benchmarking activities and reporting activities was clearly a win. Using a database to connect the activities created the possibility of using other analysis programs besides our own. Conversely, the report generator does not depend on Q-bench for its data, but can read any column-oriented data file. Thus VIEW projections can be used for evaluating most any performance-related data.

Second, good software doesn't HAVE to prompt. We think this project goes a long way to simplifying benchmarks and yet is straightforward to use.

Furthermore, pleasant surprizes can happen, even in fairly cut and dried benchmark software projects. Several people have remarked Kiviat diagrams approach "art" in their own right. (Of course, we feel they convey somewhat more information than just that.) Kiviat diagrams were not even supposed to "work" on line-printer plots, as printer resolution was deemed inadequate.

Pleasant diversions can also happen, even in cut and dried projects. We will only issue the challenge here, the solution will be discussed at the talk.

### "CHECKSUMMED HEADER" PUZZLE

It is deemed essential to put a small header file at the head of a certain unnamed company's software packages. The header file must say something to the effect, "/* serial no. XXX, checksum=YYY */".

> Furthermore, if anyone checksums such a program file (including its header) it must match the value YYY as announced in its header. How would you create the header files?

Finally, we must conclude whether or not the project goals were met. One goal was to automate the benchmarking process for non-experts. We believe we succeeded. Non-experts can compare machine performance in terms of "word-processing", "data-base operations", "compiling", etc. with as much relevance as experts can compare "disk-thruput", "divide-times", and so forth.

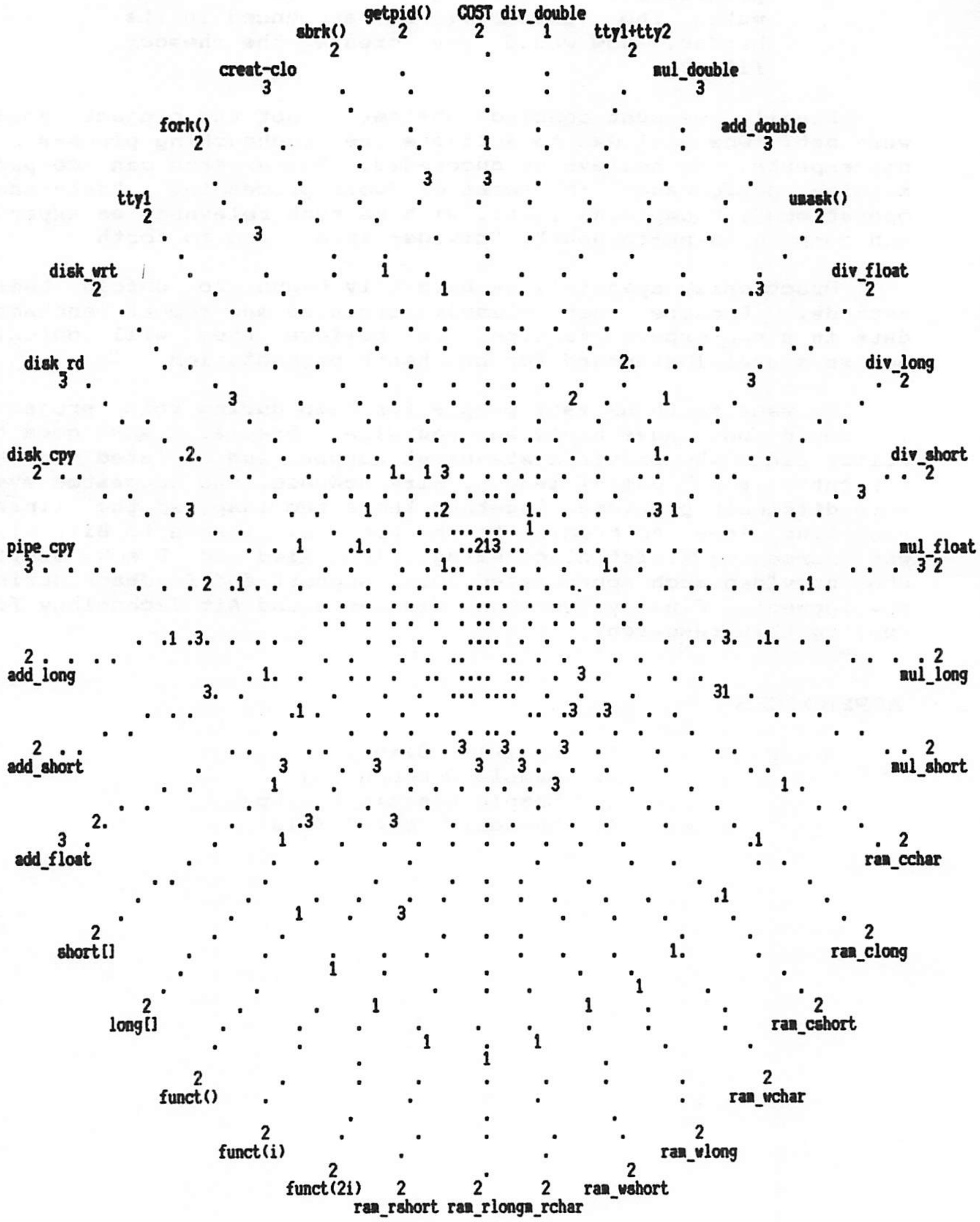
Practically speaking, we have only begun to unfold these methods. Because they cleanly organize and report benchmark data in a non-expert fashion, we believe they will quickly become a useful standard for benchmark presentation.

We want to thank many people for help during this project, it could not have happened otherwise. Special thanks goes to Walter Zintz who provided steadfast support and insisted "disk-thruput" is NOT user-friendly, Kirk McKusic, who suggested even more difficult problems. Together these two inspired the linear modelling idea so crucial to the project. Thanks to Bill King who introduced Kiviat diagramming, Linda Nied and Breck Beatie who provided much appreciated moral support and feedback during development. Finally, we thank Jim Geers and Aim Technology for funding this research.

## 6.0 APPENDICES

1. A Kiviat Diagram
2. Sample Q-bench log
3. Sample benchmark output
4. Mnemonic "defs" file

## KIVIAT DIAGRAM



legend: 1=sys a - Aim 68k 2=sys b - 68K mfr S 3=sys c - 68K mfr D

## SAMPLE Q-BENCH RUNNING COMMENTARY

```
What is  this machine's name: labrat
What is its price in dollars: 23000


Testing Started.
TEST00 labrat                      23000.
TEST01 getpid() calls               1824:    2 kcalls/sec or  548 microsec/call
TEST02 sbrk(0) calls                 796:  796 calls/sec or 1256 microsec/call
TEST03 creat-close calls              25:   25 pairs/sec or 40000 microsec/pair
TEST33 umask(0) calls               1480:    1 kcalls/sec or  676 microsec/call
TEST04 process forks                   5:    5 forks/sec or 200000 microsec/fork
TEST05 tty1 write                    696:  696 bytes/sec or 1437 microsec/byte
TEST36 tty1+tty2 write               787:  787 bytes/sec or 1271 microsec/byte
TEST06 disk write                  19765:   20 kbytes/sec or   51 microsec/byte
TEST07 disk read                   22016:   22 kbytes/sec or   45 microsec/byte
TEST08 disk copy                    9506:   10 kbytes/sec or  105 microsec/byte
TEST09 pipe copy                   70035:   70 kbytes/sec or   14 microsec/byte
TEST10 add LONG                   640000:  640 kadds/sec or    2 microsec/add
TEST11 add SHORT                  205398:  205 kadds/sec or    5 microsec/add
TEST12 add FLOAT                    1673:    2 kadds/sec or  598 microsec/add
TEST34 add DOUBLE                   1684:    2 kadds/sec or  594 microsec/add
TEST13 array ref short[short]     129431:  129 krefs/sec or    8 microsec/ref
TEST14 array ref long[long]       140502:  141 krefs/sec or    7 microsec/ref
TEST15 call funct()                48852:   49 kcalls/sec or   20 microsec/call
TEST16 call funct(int)             33903:   34 kcalls/sec or   29 microsec/call
TEST17 call funct(int,int)         26852:   27 kcalls/sec or   37 microsec/call
TEST18 ram read SHORT            1170753: 1171 kbytes/sec or  854 nanosec/byte
TEST19 ram read LONG             1662694: 1663 kbytes/sec or  601 nanosec/byte
TEST20 ram read CHAR              593068:  593 kbytes/sec or    2 microsec/byte
TEST21 ram write SHORT            702452:  702 kbytes/sec or    1 microsec/byte
TEST22 ram write LONG             960018:  960 kbytes/sec or    1 microsec/byte
TEST23 ram write CHAR             353932:  354 kbytes/sec or    3 microsec/byte
TEST24 ram copy SHORT             797932:  798 kbytes/sec or    1 microsec/byte
TEST25 ram copy LONG              990988:  991 kbytes/sec or    1 microsec/byte
TEST26 ram copy CHAR              404216:  404 kbytes/sec or    2 microsec/byte
TEST27 multiply SHORT              20540:   21 kmults/sec or   49 microsec/mult
TEST28 multiply LONG               19640:   20 kmults/sec or   51 microsec/mult
TEST29 multiply FLOAT               1759:    2 kmults/sec or  569 microsec/mult
TEST35 multiply DOUBLE              1700:    2 kmults/sec or  588 microsec/mult
TEST30 divide SHORT                12891:   13 kdivs/sec or   78 microsec/div
TEST31 divide LONG                 11218:   11 kdivs/sec or   89 microsec/div
TEST32 divide FLOAT                  931:  931 divs/sec or 1074 microsec/div
TEST37 divide DOUBLE                 931:  931 divs/sec or 1074 microsec/div
Testing Completed.
```

## SAMPLE MACHINE REPORT

**ARITHMETIC INSTRUCTION TIMES (microseconds per op)**

| | short | long | float | double |
|---|---|---|---|---|
| + add | 5 | 2 | 598 | 594 |
| * multiply | 49 | 51 | 569 | 588 |
| / divide | 78 | 89 | 1074 | 1074 |

**MEMORY LOOP ACCESS TIMES (microseconds per byte)**

| | read | write | copy |
|---|---|---|---|
| CHAR type | 2 | 3 | 2 |
| SHORT type | 854ns | 1 | 1 |
| LONG type | 601ns | 1 | 1 |

**INPUT/OUTPUT RATES (bytes/sec)**

| | read | write | copy |
|---|---|---|---|
| DISK | 22k | 20k | 10k |
| PIPE | | | 70k |
| TTY 1 | | 696 | |
| TTY 1+2 | | 787 | |
| RAM 1-byte | | | 404k |
| RAM 4-byte | | | 991k |

**ARRAY SUBSCRIPT REFERENCES (microseconds)**

| short[] | long[] |
|---|---|
| 8 | 7 |

**FUNCTION REFERENCES (microseconds/ref)**

| 0-parameters | 1-parameter | 2-parameters |
|---|---|---|
| funct() | funct(i) | funct(i,i) |
| 20 | 29 | 37 |

**PROCESS FORKS**
( 39k bytes)
5 per second

**SYSTEM KERNEL CALLS** (calls-per-second and microseconds per call)

| | | |
|---|---|---|
| getpid() calls: | 2 kcalls/sec or | 548 microseconds/call |
| sbrk(0) calls: | 796 calls/sec or | 1256 microseconds/call |
| create/close calls: | 25 pairs/sec or | 40000 microseconds/pair |
| umask(0) calls: | 1 kcalls/sec or | 676 microseconds/call |

## MNEMONIC DEFINITIONS FILE

```
all     all 50 tests from qbench
all     1,1,getpid(),syscall,sec 1,2,sbrk(),syscall,sec
all     1,3,creat-clo,file,sec   1,4,fork(),fork,sec
all     1,5,tty1,byte,sec        1,6,disk_wrt,byte,sec
all     1,7,disk_rd,byte,sec     1,8,disk_cpy,byte,sec
all     1,9,pipe_cpy,byte,sec    1,10,add_long,add,sec
all     1,11,add_short,add,sec   1,12,add_float,add,sec
all     1,13,short[],ref,sec     1,14,long[],ref,sec
all     1,15,funct(),call,sec    1,16,funct(i),call,sec
all     1,17,funct(2i),call,sec  1,18,ram_rshort,byte,sec
all     1,19,ram_rlong,byte,sec  1,20,ram_rchar,byte,sec
all     1,21,ram_wshort,byte,sec 1,22,ram_wlong,byte,sec
all     1,23,ram_wchar,byte,sec  1,24,ram_cshort,byte,sec
all     1,25,ram_clong,byte,sec  1,26,ram_cchar,byte,sec
all     1,27,mul_short,mult,sec  1,28,mul_long,mult,sec
all     1,29,mul_float,mult,sec  1,30,div_short,div,sec
all     1,31,div_long,div,sec    1,32,div_float,div,sec
all     1,33,umask(),call,sec
all     1,34,add_double,add,sec  1,35,mul_double,mult,sec
all     1,36,tty1+tty2,byte,sec
all     1,37,div_double,div,sec

ram     10,20 10,18 10,19 10,20 10,21 10,22 10,23 10,24 10,25 10,26 -ram
math    16,27 16,28 20,11 16,10 16,30 16,31    - math = integer  + * /
float   20,12 16,29 16,32 16,34 16,35 16,37    - float = flt+dbl  + * /
disk    25,03 25,06 25,07 25,08                - disk = cre/clo+r/w/copy
tty     51,05 49,36 - ttys = one and two-write
pipe    100,09      - pipe = pipe copy
logic   34,15 33,16 33,17                      - logic = function calls

wp      word-processing -       Much string ram, disk and tty.
wp      32,20  1,15  1,27  1,29  32,8  1,3  1,9  32,5

db      data-base operations -  Much ram, logic, disk, and file ops.
db      24,20  24,15  1,27  1,29  24,8  24,3  1,9  1,5

sp      spread-sheet operations - Much ram, logic, math and tty.
sp       1,20  24,15  24,27  24,29  1,8  1,3  1,9  24,5

sc      scientific calculations - Much ram, logic, math, disk and tty.
sc       1,20  24,15  24,27  24,29  24,8  1,3  1,9  1,5

ac      accounting operations - Much logic and math.
ac       1,20  19,15  19,27  19,29  21,8  19,3  1,9  1,5

co      compiling and language development - Much logic, integer math,
co                disk, and file-ops.
co       1,20  24,15  24,27  1,29  24,8  24,3  1,9  1,5

gr      graphics operations -  Much ram, logic, integer math.
gr      32,20  32,15  32,27  1,29  1,8  1,3  1,9  1,5

us      user-friendly interface operations - Much ram, file-ops, pipes,
us               and tty.
us      24,20  1,15  1,27  1,29  1,8  24,3  24,9  24,5
```